Exhibit 14 to Complaint
Intellectual Ventures I LLC and Intellectual Ventures II LLC

Example Southwest Count 6 Systems and Services
U.S. Patent No. 7,257,582 ("'582 Patent")

The Accused Systems and Services include without limitation Southwest systems and services that utilize Spark; all past, current, and future systems and services that operate in the same or substantially similar manner as the specifically identified systems and services; and all past, current, and future Southwest systems and services that have the same or substantially similar features as the specifically identified systems and services ("Example Southwest Count 6 Systems and Services" or "Southwest Systems and Services").[1]

On information and belief, the Southwest Systems and Services use Spark in its private cloud(s). For example, Southwest posts, or has posted, job opportunities that require familiarity with Spark technology.

- *See* https://www.linkedin.com/in/meghanaa-malleboina/, job profile of data engineer, stating that they utilized Pyspark for distributed data processing, leveraging RDD and DataFrame APIs for large-scale data transformations and implementing window functions for complex aggregations. (last accessed 10/8/24).
- *See* https://www.linkedin.com/in/sudhir-saxena-45b9a524/, job profile of senior data engineer stating that they ued PySpark to create source to target mapping to design Data pipeline. (last accessed 10/8/24).
- *See* https://www.linkedin.com/in/girishkumar81/, job profile of senior Azure data engineer who uses Apache Spark for his position at Southwest. (last accessed 10/8/24).
- *See* https://www.linkedin.com/in/amaranath-repati-a120bb22b/, job profile of data engineer stating the use of Spark. (last accessed 10/8/24).
- *See* https://www.linkedin.com/in/shreeja-srinivas-444961179, job profile of data engineer stating the use of Spark. (last accessed 10/8/24).

As another example, Southwest has stated that it is investing in cloud technology and has "moved about 50% of its technology" to the cloud and has indicated cloud migration is one of its areas of focus for 2024 and beyond. Source: https://www.phocuswire.com/southwest-airlines-cio-tech-investment.

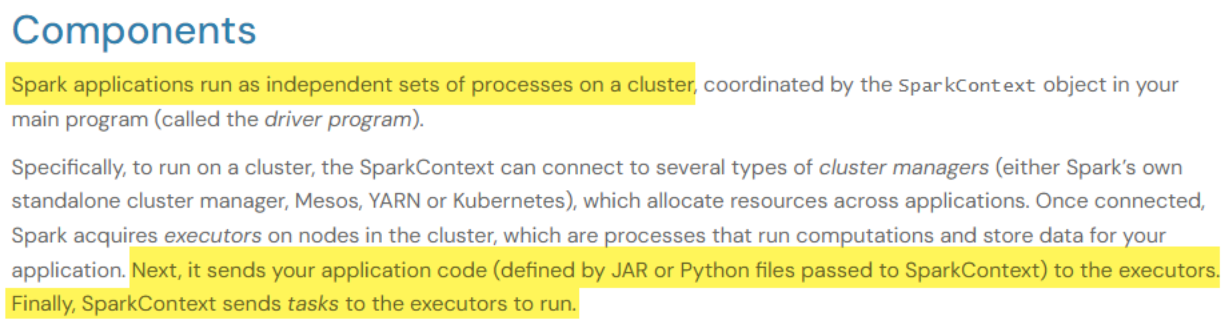
On information and belief, other information confirms Southwest uses Spark technology.

[1] For the avoidance of doubt, Plaintiffs do not accuse public clouds of Southwest if those services are provided by a cloud provider with a license to Plaintiffs' patents that covers Southwest's activities. IV will provide relevant license agreements for cloud providers in discovery. To the extent any of these licenses are relevant to Southwest's activities, Plaintiffs will meet and confer with Southwest about the impact of such license(s).

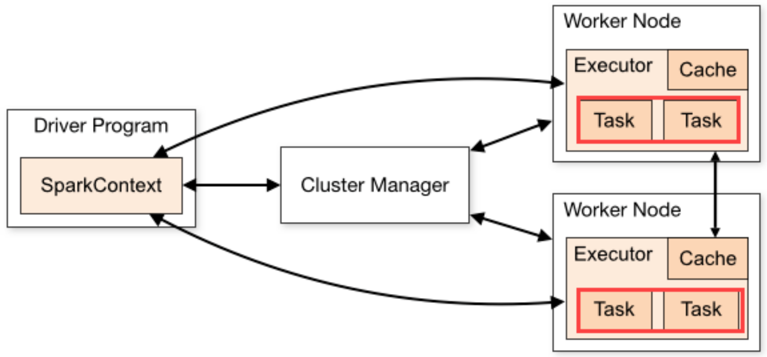
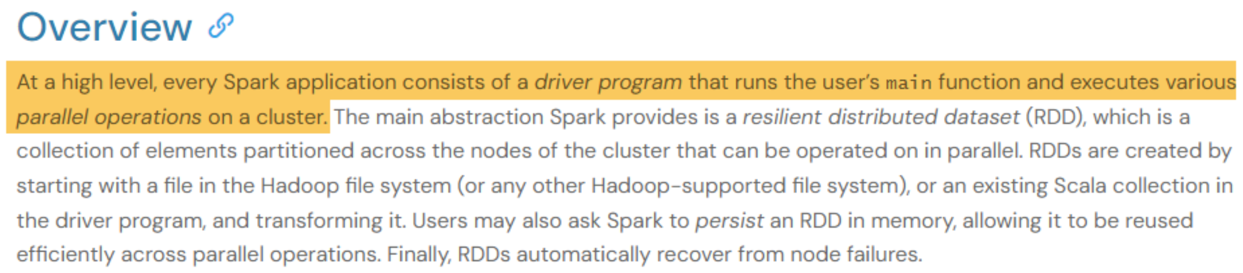
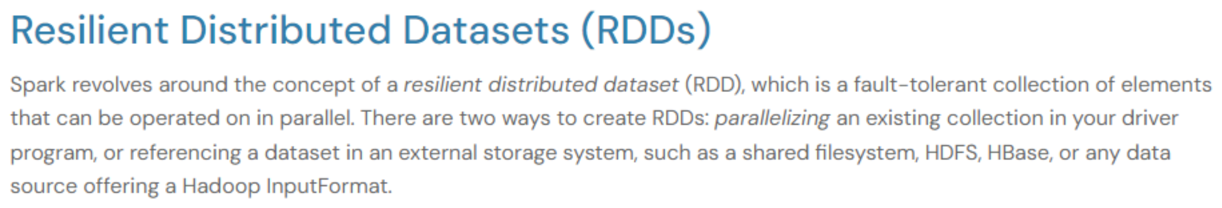
A friend of mine recently had a data engineering interview at Southwest Airlines. The interview focused on real-world scenarios, especially working with large datasets. One question he faced was particularly challenging and required a good understanding of flight data analysis.

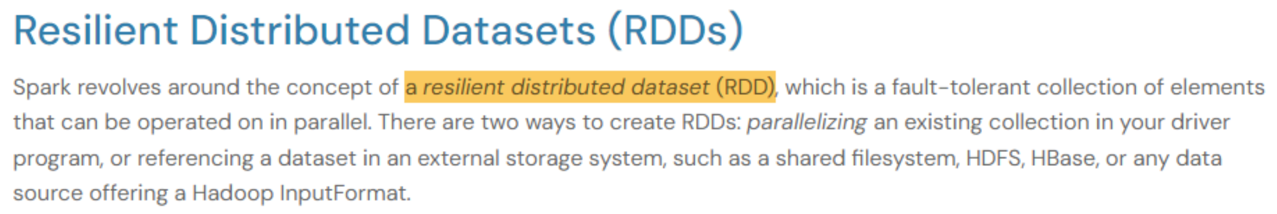
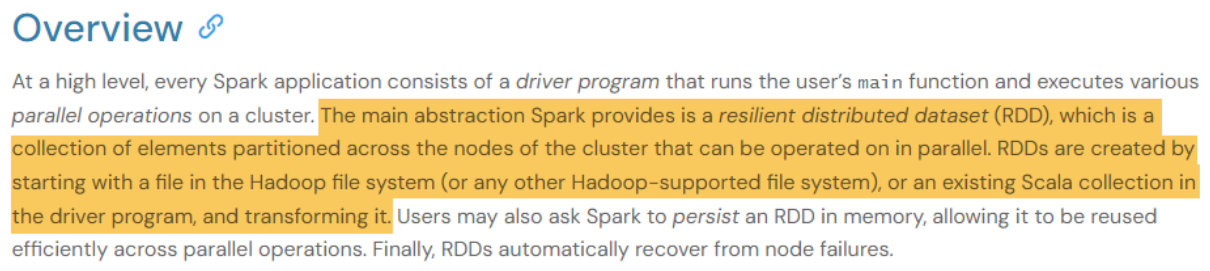
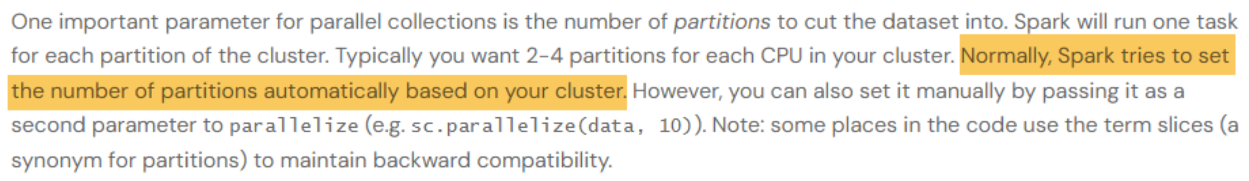
Source: https://medium.com/towards-data-engineering/inside-a-southwest-airlines-data-engineering-interview-2f33ed964b5c. [2]

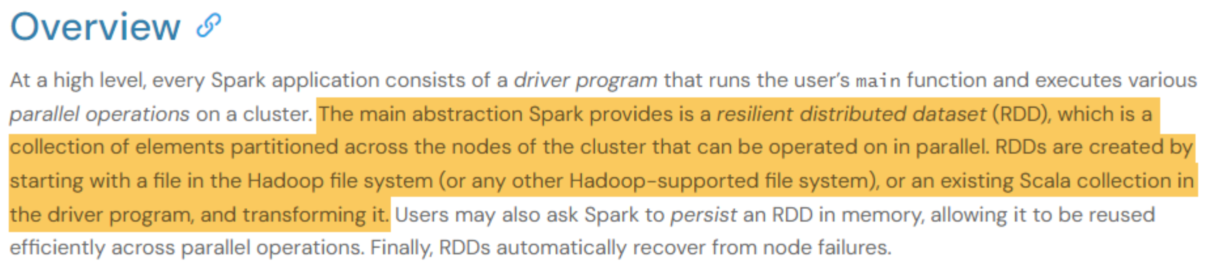
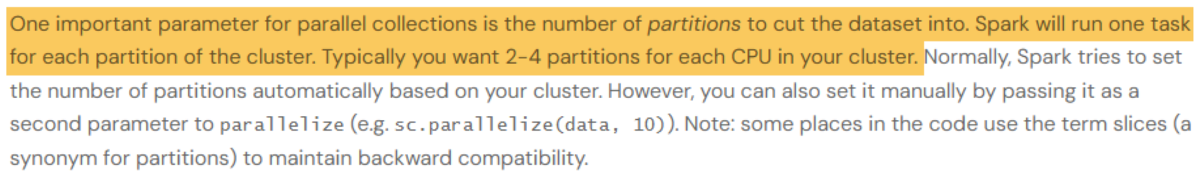
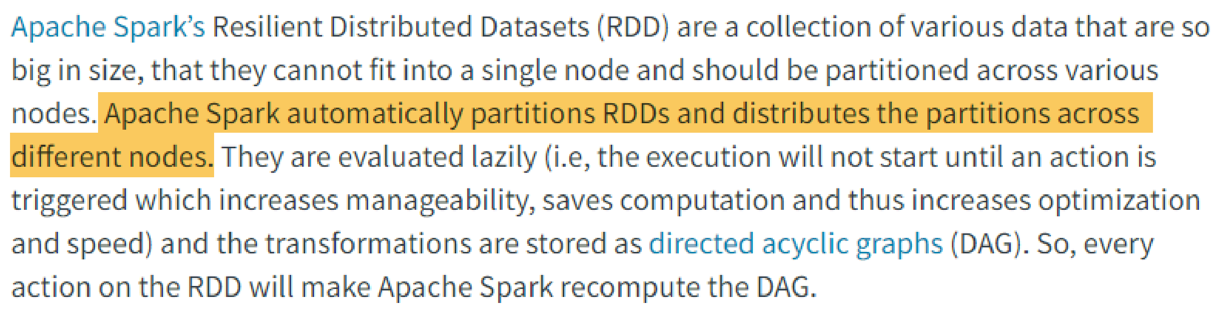
[2] All sources cited in this document were publicly accessible as of the filing date of the Complaint.

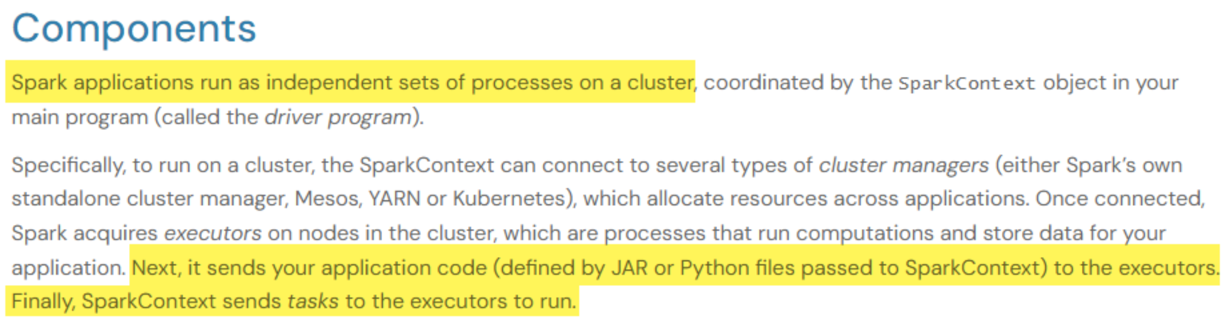
U.S. Patent No. 7,257,582 (Claim 1)	
Claim(s)	**Example Southwest Count 6 Systems and Services**
1. A method of effecting on a preexisting input file a computer-executable process comprised of a plurality of subtasks, the method comprising the steps of:	To the extent this preamble is limiting, on information and belief, the Southwest Count 6 Systems and Services practice a method of effecting on a preexisting input file a computer-executable process comprised of a plurality of subtasks. Spark includes a Cluster Mode where it connects to multiple cluster managers that allocate resources across applications. For example, Spark uses executors on nodes in the clusters to run tasks corresponding to an input file/data set that has been transformed into a resilient distributed dataset (RDD).  Components Spark applications run as independent sets of processes on a cluster, coordinated by the `SparkContext` object in your main program (called the *driver program*). Specifically, to run on a cluster, the SparkContext can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, Mesos, YARN or Kubernetes), which allocate resources across applications. Once connected, Spark acquires *executors* on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends *tasks* to the executors to run. Source: https://spark.apache.org/docs/latest/cluster-overview.html.[3]

[3] Annotations added unless otherwise noted.

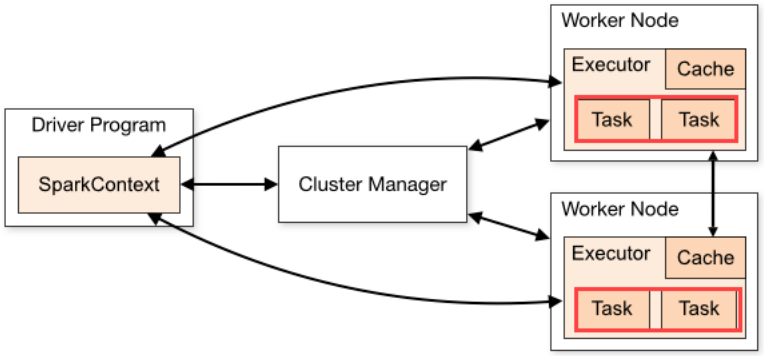
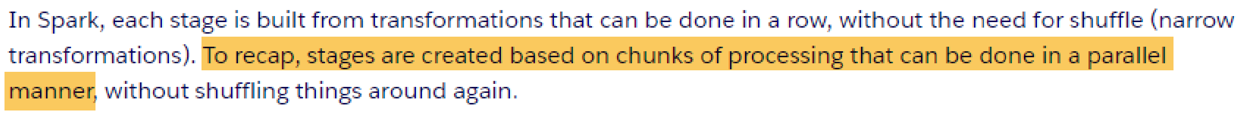
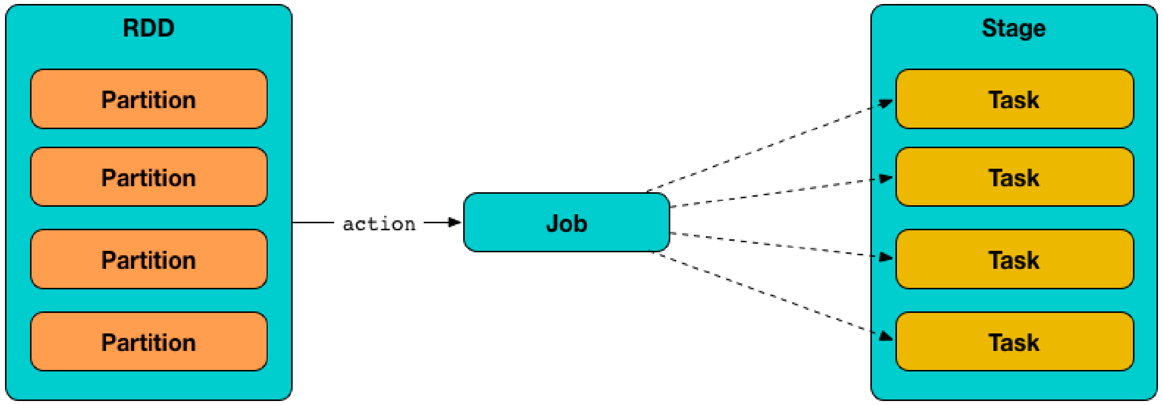
U.S. Patent No. 7,257,582 (Claim 1)	
Claim(s)	**Example Southwest Count 6 Systems and Services**
	 Driver Program SparkContext Cluster Manager Worker Node Executor Cache Task Task Worker Node Executor Cache Task Task Source: https://spark.apache.org/docs/latest/cluster-overview.html.  Overview At a high level, every Spark application consists of a *driver program* that runs the user's `main` function and executes various *parallel operations* on a cluster. The main abstraction Spark provides is a *resilient distributed dataset* (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to *persist* an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.  Resilient Distributed Datasets (RDDs) Spark revolves around the concept of a *resilient distributed dataset* (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs: *parallelizing* an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.

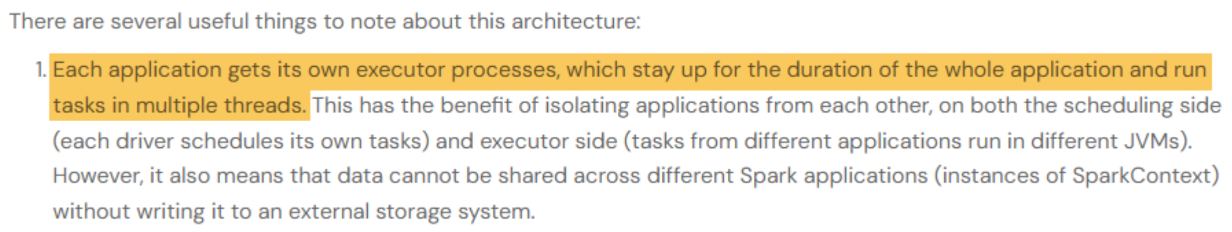
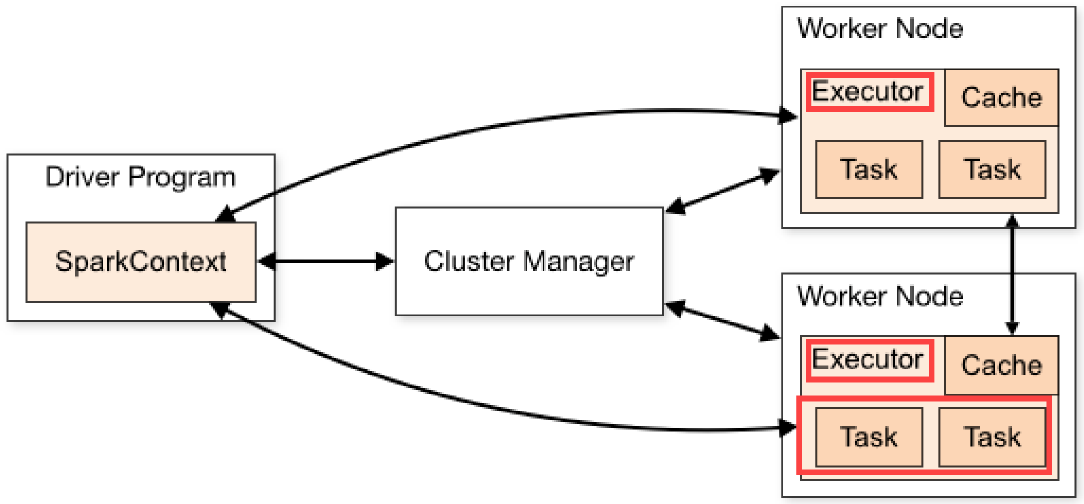
U.S. Patent No. 7,257,582 (Claim 1)	
Claim(s)	**Example Southwest Count 6 Systems and Services**
(a) automatically determining file allocation and logically subdividing records of said input file into a plurality of partitions;	On information and belief, the Southwest Count 6 Systems and Services practice automatically determining file allocation and logically subdividing records of said input file into a plurality of partitions. Spark creates an RDD from datasets including files. This process includes forming a logical division of the datasets. An RDD results in an automatic file allocation across multiple nodes in a cluster, so that the nodes of the cluster can operate on the RDD in parallel.  Resilient Distributed Datasets (RDDs) Spark revolves around the concept of *a resilient distributed dataset* (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs: *parallelizing* an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.  Overview At a high level, every Spark application consists of a *driver program* that runs the user's `main` function and executes various *parallel operations* on a cluster. The main abstraction Spark provides is a *resilient distributed dataset* (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to *persist* an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.  One important parameter for parallel collections is the number of *partitions* to cut the dataset into. Spark will run one task for each partition of the cluster. Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to `parallelize` (e.g. `sc.parallelize(data, 10)`). Note: some places in the code use the term slices (a synonym for partitions) to maintain backward compatibility. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.

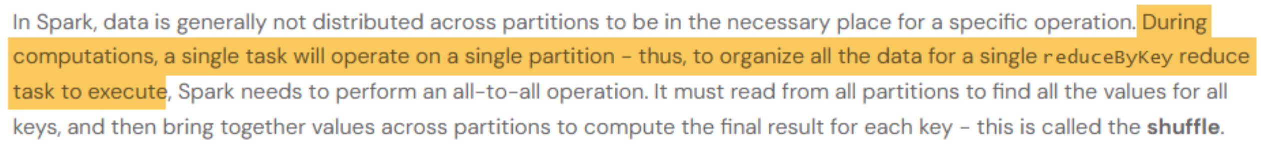
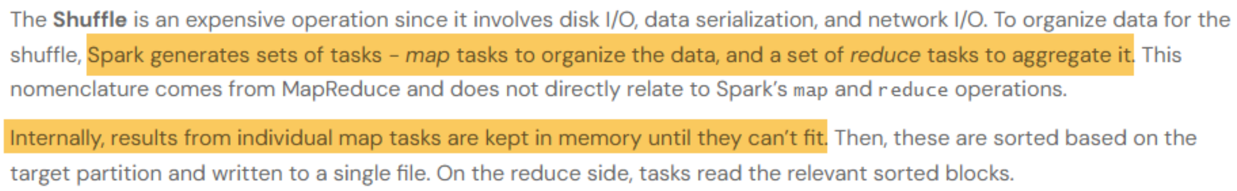
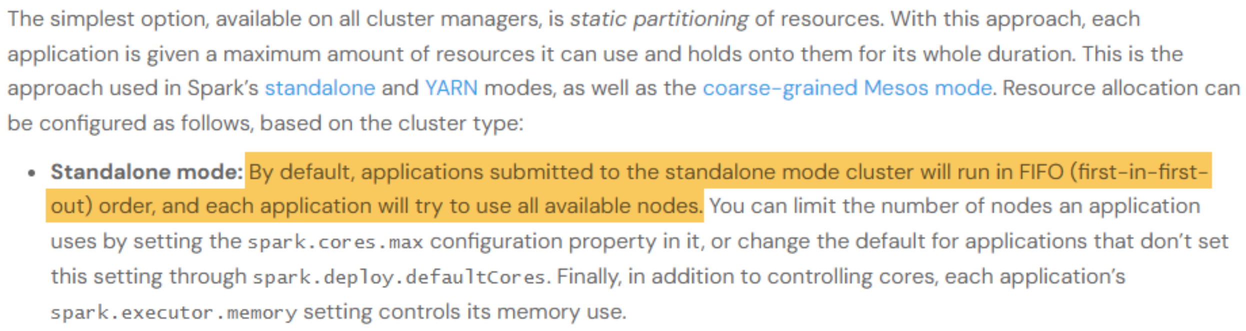
U.S. Patent No. 7,257,582 (Claim 1)	
Claim(s)	**Example Southwest Count 6 Systems and Services**
(b) distributing descriptions of all of said partitions to each of a plurality of subtask processors	On information and belief, the Southwest Count 6 Systems and Services practice distributing descriptions of all of said partitions to each of a plurality of subtask processors. Spark distributes partitions to clusters which includes multiple machines or nodes for processing. For example, Spark prepares jobs to perform work on the partitions. Jobs are divided into stages, and stages are divided into tasks, with one task assigned for each partition.  Overview At a high level, every Spark application consists of a *driver program* that runs the user's `main` function and executes various *parallel operations* on a cluster. The main abstraction Spark provides is a *resilient distributed dataset* (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to *persist* an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.  One important parameter for parallel collections is the number of *partitions* to cut the dataset into. Spark will run one task for each partition of the cluster. Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to `parallelize` (e.g. `sc.parallelize(data, 10)`). Note: some places in the code use the term slices (a synonym for partitions) to maintain backward compatibility. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.  Apache Spark's Resilient Distributed Datasets (RDD) are a collection of various data that are so big in size, that they cannot fit into a single node and should be partitioned across various nodes. Apache Spark automatically partitions RDDs and distributes the partitions across different nodes. They are evaluated lazily (i.e, the execution will not start until an action is triggered which increases manageability, saves computation and thus increases optimization and speed) and the transformations are stored as directed acyclic graphs (DAG). So, every action on the RDD will make Apache Spark recompute the DAG.

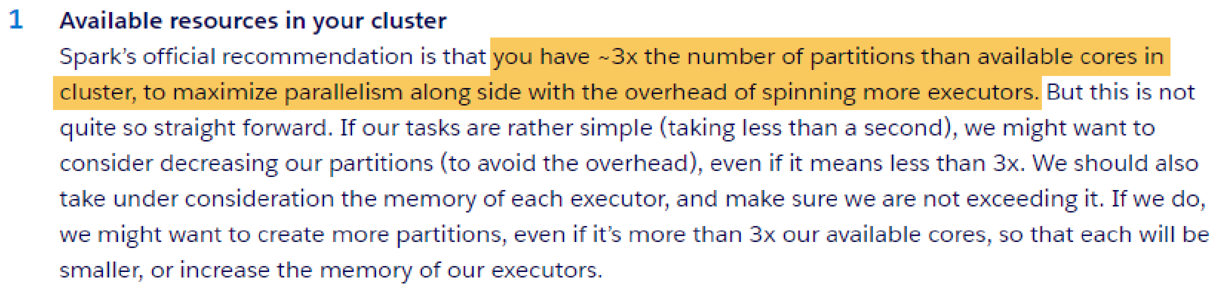
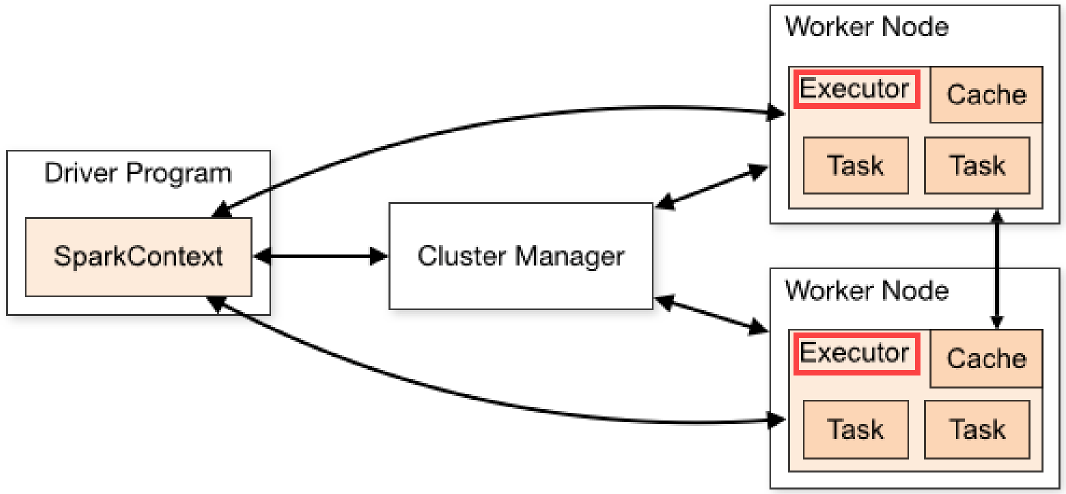
U.S. Patent No. 7,257,582 (Claim 1)	
Claim(s)	**Example Southwest Count 6 Systems and Services**
	Source: https://www.talend.com/resources/intro-apache-spark-partitioning/.
(c) simultaneously executing at least a respective one of the subtasks of the computer-executable process in each of at least some of said processors on a respective one of the partitions with each subtask reading and processing the respective partition so as to process the respective partition and produce respective subtask output and;	On information and belief, the Southwest Count 6 Systems and Services practice simultaneously executing at least a respective one of the subtasks of the computer-executable process in each of at least some of said processors on a respective one of the partitions with each subtask reading and processing the respective partition so as to process the respective partition and produce respective subtask output. Spark worker nodes simultaneously execute at least one task relating to the partition.  Components Spark applications run as independent sets of processes on a cluster, coordinated by the `SparkContext` object in your main program (called the *driver program*). Specifically, to run on a cluster, the SparkContext can connect to several types of *cluster managers* (either Spark's own standalone cluster manager, Mesos, YARN or Kubernetes), which allocate resources across applications. Once connected, Spark acquires *executors* on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends *tasks* to the executors to run. Source: https://spark.apache.org/docs/latest/cluster-overview.html.[4]

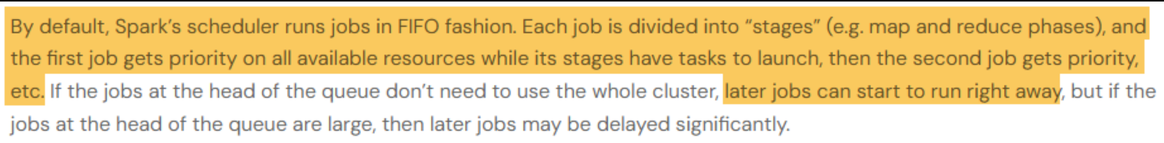
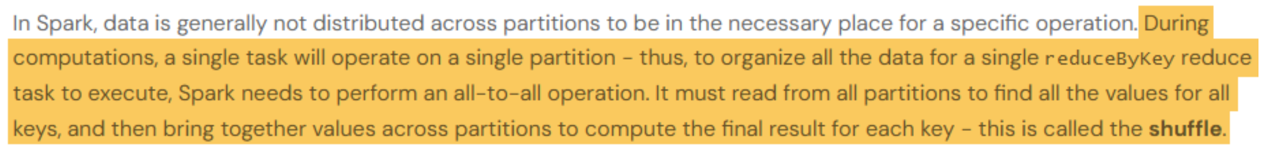
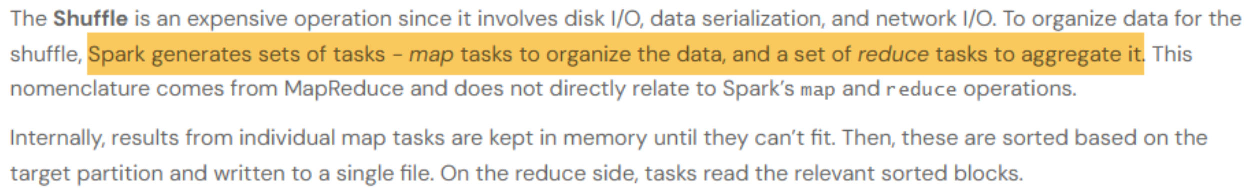
[4] Annotations added unless otherwise noted.

U.S. Patent No. 7,257,582 (Claim 1)	
Claim(s)	**Example Southwest Count 6 Systems and Services**
	 Driver Program SparkContext Cluster Manager Worker Node Executor Cache Task Task Worker Node Executor Cache Task Task Source: https://spark.apache.org/docs/latest/cluster-overview.html. In Spark, each stage is built from transformations that can be done in a row, without the need for shuffle (narrow transformations). To recap, stages are created based on chunks of processing that can be done in a parallel manner, without shuffling things around again.  RDD Partition Partition Partition Partition action Job Stage Task Task Task Task

U.S. Patent No. 7,257,582 (Claim 1)	
Claim(s)	**Example Southwest Count 6 Systems and Services**
	Source: https://engineering.salesforce.com/how-to-optimize-your-apache-spark-application-with-partitions-257f2c1bb414/. Spark's tasks relating to each partition are being executed in parallel by the worker nodes.  There are several useful things to note about this architecture: 1. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads. This has the benefit of isolating applications from each other, on both the scheduling side (each driver schedules its own tasks) and executor side (tasks from different applications run in different JVMs). However, it also means that data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system. Source: https://spark.apache.org/docs/latest/cluster-overview.html.  Driver Program SparkContext Cluster Manager Worker Node Executor Cache Task Task Worker Node Executor Cache Task Task

U.S. Patent No. 7,257,582 (Claim 1)	
Claim(s)	**Example Southwest Count 6 Systems and Services**
	 In Spark, data is generally not distributed across partitions to be in the necessary place for a specific operation. During computations, a single task will operate on a single partition – thus, to organize all the data for a single `reduceByKey` reduce task to execute, Spark needs to perform an all-to-all operation. It must read from all partitions to find all the values for all keys, and then bring together values across partitions to compute the final result for each key – this is called the **shuffle**. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.  The **Shuffle** is an expensive operation since it involves disk I/O, data serialization, and network I/O. To organize data for the shuffle, Spark generates sets of tasks – *map* tasks to organize the data, and a set of *reduce* tasks to aggregate it. This nomenclature comes from MapReduce and does not directly relate to Spark's `map` and `reduce` operations. Internally, results from individual map tasks are kept in memory until they can't fit. Then, these are sorted based on the target partition and written to a single file. On the reduce side, tasks read the relevant sorted blocks. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.
(d) thereafter repeating step (c) in at least some of the subtask processors each with another unprocessed partition on a first-come/first-served basis; and	On information and belief, the Southwest Count 6 Systems and Services practice repeating step (c) in at least some of the subtask processors each with another unprocessed partition on a first-come/first-served basis. Spark task output produced by each worker node is combined to produce processed output for the related partition.  The simplest option, available on all cluster managers, is *static partitioning* of resources. With this approach, each application is given a maximum amount of resources it can use and holds onto them for its whole duration. This is the approach used in Spark's standalone and YARN modes, as well as the coarse-grained Mesos mode. Resource allocation can be configured as follows, based on the cluster type: • **Standalone mode:** By default, applications submitted to the standalone mode cluster will run in FIFO (first-in-first-out) order, and each application will try to use all available nodes. You can limit the number of nodes an application uses by setting the `spark.cores.max` configuration property in it, or change the default for applications that don't set this setting through `spark.deploy.defaultCores`. Finally, in addition to controlling cores, each application's `spark.executor.memory` setting controls its memory use. Source: https://spark.apache.org/docs/latest/job-scheduling.html.

U.S. Patent No. 7,257,582 (Claim 1)	
Claim(s)	**Example Southwest Count 6 Systems and Services**
	 1 **Available resources in your cluster** Spark's official recommendation is that you have ~3x the number of partitions than available cores in cluster, to maximize parallelism along side with the overhead of spinning more executors. But this is not quite so straight forward. If our tasks are rather simple (taking less than a second), we might want to consider decreasing our partitions (to avoid the overhead), even if it means less than 3x. We should also take under consideration the memory of each executor, and make sure we are not exceeding it. If we do, we might want to create more partitions, even if it's more than 3x our available cores, so that each will be smaller, or increase the memory of our executors. Source: https://engineering.salesforce.com/how-to-optimize-your-apache-spark-application-with-partitions-257f2c1bb414/.  Worker Node Executor Cache Task Task Driver Program SparkContext Cluster Manager Worker Node Executor Cache Task Task Source: https://spark.apache.org/docs/latest/cluster-overview.html.

U.S. Patent No. 7,257,582 (Claim 1)	
Claim(s)	**Example Southwest Count 6 Systems and Services**
	 By default, Spark's scheduler runs jobs in FIFO fashion. Each job is divided into "stages" (e.g. map and reduce phases), and the first job gets priority on all available resources while its stages have tasks to launch, then the second job gets priority, etc. If the jobs at the head of the queue don't need to use the whole cluster, later jobs can start to run right away, but if the jobs at the head of the queue are large, then later jobs may be delayed significantly. Source: https://spark.apache.org/docs/latest/job-scheduling.html.
(e) generating at least one output combining all of the subtask outputs and reflecting the processing of all of said subtasks.	On information and belief, the Southwest Count 6 Systems and Services practice generating at least one output combining all of the subtask outputs and reflecting the processing of all of said subtasks. Spark task output produced by each worker node is combined to produce final processed output for the corresponding partition.  In Spark, data is generally not distributed across partitions to be in the necessary place for a specific operation. During computations, a single task will operate on a single partition – thus, to organize all the data for a single `reduceByKey` reduce task to execute, Spark needs to perform an all-to-all operation. It must read from all partitions to find all the values for all keys, and then bring together values across partitions to compute the final result for each key – this is called the **shuffle**. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.  The **Shuffle** is an expensive operation since it involves disk I/O, data serialization, and network I/O. To organize data for the shuffle, Spark generates sets of tasks – *map* tasks to organize the data, and a set of *reduce* tasks to aggregate it. This nomenclature comes from MapReduce and does not directly relate to Spark's `map` and `reduce` operations. Internally, results from individual map tasks are kept in memory until they can't fit. Then, these are sorted based on the target partition and written to a single file. On the reduce side, tasks read the relevant sorted blocks. Source: https://spark.apache.org/docs/latest/rdd-programming-guide.html.